PFEM 3D code structure

Last updated on Saturday 6th February, 2021

Simon FÉVRIER

Advisors:
Jean-Philippe PONTHOT, Romain BOMAN

LIÈGE université
Sciences Appliquées

Table of contents

Previous issues with PFEM 3D code

Issues with solvers

Supported problems:

1) Conduction problem (implicit scheme),
2) Boussinesq problem (implicit and explicit scheme),
3) Newtonian flow problem (implicit and explicit scheme).

Issues : copy-pasted code between the different solvers (because lack of time to do it properly before FRIA):

- matrix building code,
- Picard algorithm code,
- momentum and continuity equation code,
- etc.

NB: implicit schemes $\rightarrow$ Incompressible flow, explicit schemes $\rightarrow$ weakly-compressible flow.

Issues with extractors

Supported extractors:

1) GMSH extractor: dump states for the whole mesh to a .msh file,
2) Point extractor: extract state at a specific location,
3) MinMax extractor: extract the minimum or maximum position of the mesh,
4) Mass extractor: extract the total mass associated to a problem.

Issues: the extractor's code has to be modified for each problem/-solver that is added.
Parameters reading in json but boundary condition code in lua $\rightarrow$ unification in lua.

libpfem3DMesh

The Node class

```
class MESH_API Node
{
    ...

    private:
        Mesh* m_pMesh;
        std::array<double, 3> m_position;
        std::vector<double> m_states;
        std::vector<std::size_t> m_neighbourNodes;
        std::vector<std::size_t> m_elements;
        std::vector<std::size_t> m_facets;

        bool m_isBound = false;
        bool m_isOnFreeSurface = false;
        bool m_isFixed = false;

        int m_tag = -1;
        std::bitset<8> m_userDefFlags;

        ...

        friend class Mesh;
};
```

- `m_states`: contains the state of the fluid (ex: u, p, v, T),
- `m_neighbourNodes`: contains the index of the nodes which are in the same elements as the current node,
- `m_elements`: contains the elements which the node is inside of,
- `m_facets`: contains the boundary facets which the node is inside of,
- `m_isFixed`: the node has a velocity but do not move,
- `m_tag`: tag to obtain the name of the boundary which the node is maybe part of,
- `m_userDefFlags`: bit flags to identify which boundary condition to use.

The Element class

```
class MESH_API Element
{
    ...

    private:
        Mesh* m_pMesh;
        std::vector<std::size_t> m_nodesIndexes;
        double m_detJ;
        std::array<std::array<double, 3>, 3> m_J;
        std::array<std::array<double, 3>, 3> m_invJ;

        ...

        friend class Mesh;
};
```

- `m_nodesIndexes`: contains the index of the nodes which forms the element,
- `m_detJ`: the determinant of the change of variable to the reference space,
- `m_J`: the Jacobian matrix of the change of variable to the reference space,
- `m_invJ`: the inverse Jacobian matrix of the change of variable to the reference space.

The Facet class

```
class MESH_API Facet
{
    ...

    private:
        Mesh* m_pMesh;
        std::vector<std::size_t> m_nodesIndexes;
        std::size_t m_outNodeIndex;
        double m_detJ;
        std::array<std::array<double, 2>, 3> m_J;
        std::array<std::array<double, 3>, 2> m_invJ;

        ...

        friend class Mesh;
};
```

- `m_nodesIndexes`: contains the index of the nodes which forms the facet,
- `m_outNodeIndex`: the index of the node "in front of" the boudnary facet inside a boundary element,
- `m_detJ`: the determinant of the change of variable to the reference space,
- `m_J`: the Jacobian matrix of the change of variable to the reference space,
- `m_invJ`: the inverse Jacobian matrix of the change of variable to the reference space.

The Mesh class (1/3)

```
class MESH_API Mesh
{
    ...

    void remesh();

    private:
        double m_hchar;
        double m_alpha;
        double m_omega;
        double m_gamma;
        std::vector<double> m_boundingBox;

        bool m_computeNormalCurvature;

        unsigned short m_dim;

        std::vector<Node> m_nodesList;
        std::vector<Node> m_nodesListSave;
        std::vector<Element> m_elementsList;
        std::vector<Facet> m_facetsList;

        std::vector<std::string> m_tagNames;
        std::map<std::size_t,
                 std::array<double,3>> m_boundaryInitialPos;
        std::map<std::size_t,
                 std::array<double,3>> m_boundFSNormal;
        std::map<std::size_t,
                 double> m_freeSurfaceCurvature;

        ...
};
```

- `m_hchar`: mesh characteristic size,
- `m_alpha`: the alpha parameter of te alpha-shape,
- `m_omega`: add node if element area is greater than $\omega h_{char}^{\dim}$,
- `m_gamma`: delete node if distance is smaller than γh_{char},
- `m_boundingBox`: delete node if they are outside the bounding box,
- `m_computeNormalCurvature`: should normals and curvatures be computed while remeshing,
- `m_nodesListSave`: save of the node list,
- `m_tagNames`: name of the boundaries,
- `m_boundaryInitialPos`: initial position of nodes at the boundary,
- `m_boundFSNormal`: normal of the boundary facet for each node,
- `m_freeSurfaceCurvature`: curvature of the free surface for each node,

The Mesh class (2/3)

```
void Mesh::remesh(bool verboseOutput)
{
    checkBoundingBox(verboseOutput);
    addNodes(verboseOutput);
    removeNodes(verboseOutput);
    triangulateAlphaShape();
}
```

- `checkBoundingBox`: check if a node is outside the bounding bw,
- `addNodes`: add node in the center of element that are too big,
- `removeNodes`: remove node if it is too close from another node,
- `triangulateAlphaShape`: Delaunay triangulation and alpha-shape using CGAL.

The Mesh class (3/3)

```
typedef CGAL::Exact_predicates_inexact_constructions_kernel Kernel;
typedef Kernel::FT FT;
typedef CGAL::Triangulation_vertex_base_with_info_2<std::size_t, Kernel> Vb2;
typedef CGAL::Alpha_shape_vertex_base_2<Kernel, Vb2> asVb2;
typedef CGAL::Alpha_shape_face_base_2<Kernel> asFb2;
typedef CGAL::Triangulation_data_structure_2<asVb2,asFb2> asTds2;
typedef CGAL::Delaunay_triangulation_2<Kernel,asTds2> asTriangulation_2;
typedef CGAL::Alpha_shape_2<asTriangulation_2> Alpha_shape_2;
typedef Kernel::Point_2 Point_2; typedef Kernel::Triangle_2 Triangle_2;

void Mesh::triangulateAlphaShape2D()
{
    m_elementsList.clear(); m_facetsList.clear();

    std::vector<std::pair<Point_2, std::size_t>> pointsList;
    for(std::size_t i = 0 ; i < m_nodesList.size() ; ++i)
    {
        pointsList.push_back(std::make_pair(
                        Point_2(m_nodesList[i].m_position[0],
                                m_nodesList[i].m_position[1]), i));

        m_nodesList[i].m_isOnFreeSurface = false;
        m_nodesList[i].m_neighbourNodes.clear();
        m_nodesList[i].m_elements.clear();
        m_nodesList[i].m_facets.clear();
    }

    const Alpha_shape_2 as(pointsList.begin(), pointsList.end(),
                           m_alpha*m_alpha*m_hchar*m_hchar,
                           Alpha_shape_2::GENERAL);

    // Add elements and facets
    ...

    computeFSNormalCurvature();
}
```

- a lot of typedef to configure CGAL,
- then one call to compute the alpha-shape,
- then adding elements and facets (and be careful !).

libpfem3DSimulation

General comments on structure

The code is subdivided in three parts:

1) a `Problem` class which is responsible of holding general information about the problem being currently solved (e.g. the mesh, the current time and step, which data can be extracted, etc.),
2) a `Solver` class which is responsible of initializing the different Equation's and solving the numerical simulation for one time step,
3) a `Equation` class which is responsible of solving a particular equation.

Old parameters reading

```
{
    "ProblemType": "Conduction",
    "Remeshing": {
        "hchar": 0.5,
        "alpha": 1.2,
        "omega": 0.7,
        "gamma": 0.7,
        "boundingBox": [-1, -0.25, 5, 1.25]
    },
    "Solver": {
        "gravity": 0,
        "strongPAtFS": true,
        "Time": {
            "adaptDT": true,
            "coeffDTincrease": 1.5,
            "coeffDTdecrease": 2.0,
            "maxDT": 0.1,
            "initialDT": 0.025,
            "endTime": 6
        },
        "Fluid": {
            "rho": 1,
            "k": 237,
            "cv": 1000
        },
        "IBCs": "../../examples/2D/conduction/IBC_Incomp.lua",
        "Extractors" : [
            {
                "type": "GMSH",
                "outputFile": "results.msh",
                "timeBetweenWriting": 0.05,
                "whatToWrite": ["T"],
                "writeAs": "NodesElements"
            }
        ]
    },
    "verboseOutput": false
}
```

Json parameter file.

```
DownFixed = true
UpFixed = true
LeftFixed = true
RightFixed = true

function initStates(pos)
    return {10*math.exp(-((pos[1]-0.5)^2 + (pos[2]-0.5)^2)/0.05)}
    --return {10*math.exp(-((pos[2]-0.5)^2)/0.05)}, false
end

function DownQ(pos, initPos, t)
    return {0}
end

function RightT(pos, initPos, t)
    return {100}
end

function UpQ(pos, initPos, t)
    return {0}
end

function LeftQ(pos, initPos, t)
    return {23700.0}
end
```

Lua boundary condition file.

New parameters reading

```
Problem = {
    id = "Conduction",
    simulationTime = 1,
    verboseOutput = false,

    Mesh = {
        hchar = 0.05,
        alpha = 1.2,
        omega = 0.7,
        gamma = 0.7,
        boundingBox = {-1, -0.25, 5, 1.25},
        mshFile = "examples/2D/conduction/geometry.msh"
    },

    Extractors = {
        {
            kind = "GMSH",
            outputFile = "results.msh",
            timeBetweenWriting = 0.01,
            whatToWrite = {"T", "normals"},
            writeAs = "NodesElements"
        }
    },

    Material = {
        rho = 1,
        k = 237,
        cv = 1000
    },

    IC = {
        UpFixed = true,
        DownFixed = true,
        RightFixed = true,
        LeftFixed = true,
    },

    Solver = {
        id = "PSPG",
        adaptDT = true,
        coeffDTincrease = 1.5,
        coeffDTDecrease = 2,
        maxDT = 0.1,
        initialDT = 0.025,

        HeatEq = {
            minRes = 1e-6,
            maxIter = 10,
            BC = {

            }
        }
    }
}

function Problem.IC:initStates(pos)
    return {10*math.exp(-((pos[1]-0.5)^2 + (pos[2]-0.5)^2)/0.05)}
end

function Problem.Solver.HeatEq.BC:DownQ(pos, initPos, states, t)
    return {0, 0}
end

function Problem.Solver.HeatEq.BC:UpQ(pos, initPos, states, t)
    return {0, 0}
end

function Problem.Solver.HeatEq.BC:LeftT(pos, initPos, states, t)
    return {0}
end

function Problem.Solver.HeatEq.BC:RightT(pos, initPos, states, t)
    return {0}
end
```

Lua parameter file.

The Problem class (1/2)

```
class SIMULATION_API Problem
{
    public:
        Problem(const std::string& luaFilePath);

        ...

        virtual void displayParams() const;
        std::string getID() const noexcept;

        virtual std::vector<std::string>
        getWrittableDataName() const;

        virtual std::vector<double>
        getWrittableData(const std::string& name,
                         std::size_t nodeIndex) const;

        virtual std::vector<std::string>
        getGlobalWrittableDataName() const;

        virtual double
        getGlobalWrittableData(const std::string& name) const;

        std::vector<std::string>
        getMeshWrittableDataName() const;

        std::vector<double>
        getMeshWrittableData(const std::string& name,
                             std::size_t nodeIndex) const;

        void simulate();
        void updateTime(double timeStep);

        ...
};
```

- parameters loaded from lua file,
- `getWrittableDataName`: get the name of the data which can be written,
- `getGlobalWrittableDataName`: get the name of the global data which can be written (e.g.: mass),
- `getMeshWrittableDataName`: get the name of the mesh data which can be written (e.g.: normals, curvatures, debug),
- `simulate`: run the simuaion from $t = 0$ to $t = t_{max}$,
- `updateTime`: update the time of the simulation (used by the solver).

The Problem class (2/2)

```
class SIMULATION_API Problem
{
    public:
        ...

    protected:
        std::string m_id;
        double m_time;
        double m_maxTime;
        std::size_t m_step;
        unsigned int m_statesNumber;

        bool m_verboseOutput;

        unsigned int m_nThreads;
        std::vector<SolTable> m_problemParams;

        std::unique_ptr<Mesh> m_pMesh;
        std::unique_ptr<Solver> m_pSolver;
        std::vector<std::unique_ptr<Extractor>> m_pExtractors;

        void addExtractors();
        void setInitialCondition();
};
```

- `m_id`: small string to parametrize the problem (IncompNoT, Boussinesq, Conduction, ...),
- `m_time`, `m_step`: current time and step,
- `m_problemParams`: vector containing the problem parameters (one per thread),
- `m_pMesh`, `m_pSolver`, `m_pExtractors`: smart pointers to the mesh, the solver and the extractors,
- `addExtractors`: helper function executed by all problems to set up the extractors,
- `setInitialCondition`: helper function executed by all problems to set up the initial condition.

The Solver class (1/3)

```
class SIMULATION_API Solver
{
    public:
        Solver(Problem* pProblem, Mesh* pMesh,
               std::vector<SolTable> problemParams);

        ...

        virtual void displayParams() const;

        bool checkBC(SolTable bcParam, unsigned int n,
                     const Node& node,
                     std::string bcString,
                     unsigned int expectedBCSize);

        virtual bool solveOneTimeStep();
        virtual void computeNextDT();

        ...

    protected:
        std::string m_id;
        std::vector<SolTable> m_solverParams;
        double m_timeStep;
        std::vector<std::unique_ptr<Equation>> m_pEquations;
        bool m_solveSucceed;
        Mesh* m_pMesh;
        Problem* m_pProblem;
};
```

- `checkBC`: helper function to check if a BC exists and return the right number of states,
- `solveOneTimeStep`: solve the numerical equations for one time step and update the mesh accordingly,
- `computeNextDT`: compute the next time step used by the solver,
- `m_id`: small string to parametrize the problem (PSPG, CDS, ...),
- `m_pEquations`: vector of smart pointers to the equations,
- `m_solveSucceed`: store if `solveOneTimeStep` was successful or not.

The Solver class (2/3)

```
if(m_pProblem->getID() == "IncompNewtonNoT")
    {
        //Only the momentum-continuity equation
        m_pEquations.resize(1);

        bcFlags = {0};
        statesIndex = {0};
        m_pEquations[0] = std::make_unique<MomContEqIncompNewton>(
            m_pProblem, this, m_pMesh, m_solverParams, materialParams,
            bcFlags, statesIndex
        );

        //Set the right node flag if the boundary condition is present
        SolTable bcParam = m_pEquations[0]->getBCParam(0);
        for(std::size_t n = 0 ; n < m_pMesh->getNodesCount() ; ++n)
        {
            const Node& node = m_pMesh->getNode(n);
            if(node.isBound())
            {
                bool res = checkBC(bcParam, n, node, "V", m_pMesh->getDim());

                if(res)
                    m_pMesh->setNodeFlag(n, 0);
            }
        }

        m_solveFunc = std::bind(&SolverIncompNewton::m_solveIncompNewtonNoT, this);
    }
```

From `SolverIncompNewton` constructor:

- `m_pEquation` is resized,
- the equations are setted up,
- the boundary conditions are checked,
- the right `solve` function is chosen.

The Solver class (3/3)

```
bool SolverIncompNewton::m_solveBoussinesq()
{
    if(m_solveHeatFirst && !m_pEquations[1]->solve())
    {
        m_solveSucceed = false;
        return m_solveSucceed;
    }

    m_solveSucceed = m_pEquations[0]->solve();
    if(!m_solveSucceed)
        return m_solveSucceed;

    if(!m_solveHeatFirst && !m_pEquations[1]->solve())
    {
        m_solveSucceed = false;
        return m_solveSucceed;
    }

    if(m_solveSucceed)
    {
        m_pProblem->updateTime(m_timeStep);
        m_pMesh->remesh(m_pProblem->isOutputVerbose());
    }

    return m_solveSucceed;
}
```

Typical `solve` function for a solver.

The Equation class

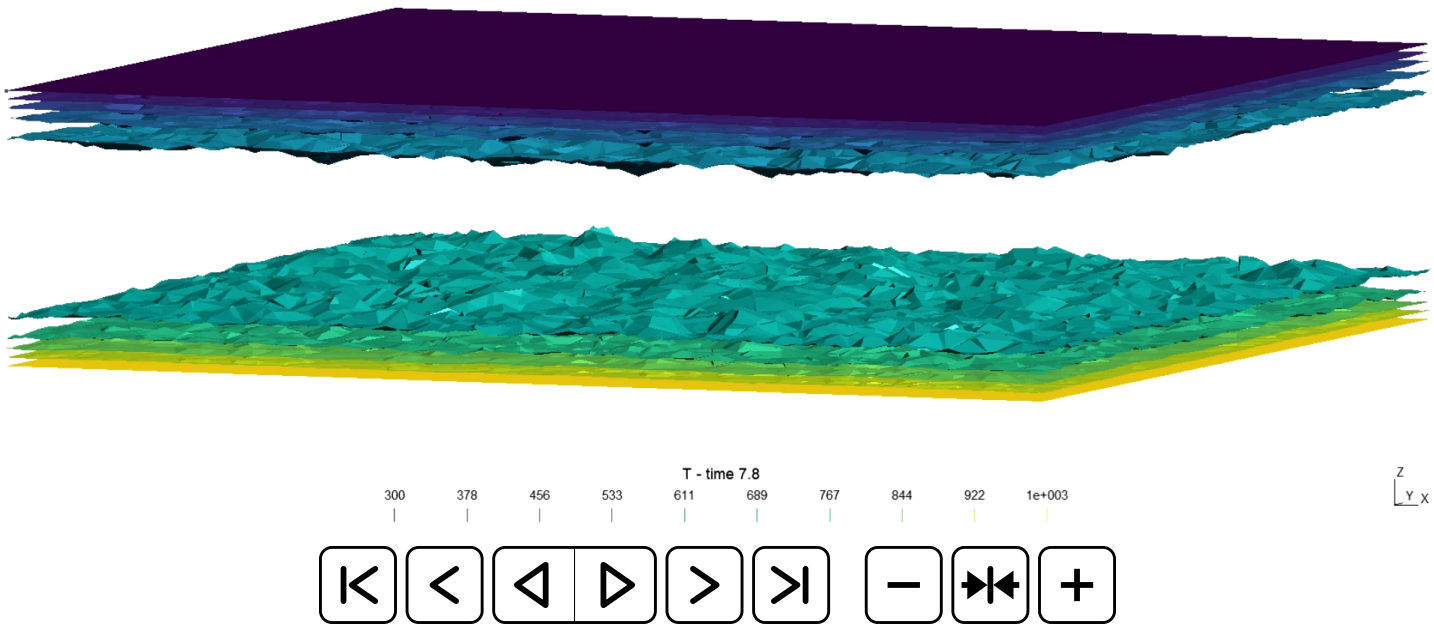
```
class SIMULATION_API Equation
{
    public:
        Equation(Problem* pProblem, Solver* pSolver,
                 Mesh* pMesh,
                 std::vector<SolTable> solverParams,
                 std::vector<SolTable> materialParams,
                 const std::vector<unsigned short>& bcFlags,
                 const std::vector<unsigned int>& statesIndex,
                 const std::string& id);
        ...

        virtual void displayParams() const;
        bool isNormalCurvNeeded() const noexcept;
        virtual double getSpeedEquiv(double he, const Node& node);
        virtual void preCompute();
        virtual bool solve();

    protected:
        bool m_needNormalCurv;
        std::string m_id;
        std::vector<SolTable> m_materialParams;
        std::vector<SolTable> m_equationParams;
        std::vector<SolTable> m_bcParams;
        std::vector<unsigned short> m_bcFlags;
        std::vector<unsigned int> m_statesIndex;
        Problem* m_pProblem;
        Solver* m_pSolver;
        Mesh* m_pMesh;
        std::unique_ptr<MatrixBuilder> m_pMatBuilder;
};
```

- `isNormalCurvNeede`: does this equation requires the computation of the normals and curvatures,
- `getSpeedEquiv`: get the equivalent numerical speed associated to this equation (if explicit),
- `preCompute`: precompute some data before solving the equation,
- `solve`: solve the equation and update the states inside the mesh
- `m_id`: small string to identify the equation in the parameter file (MomEq, MomContEq, HeatEq, ...),
- `m_bcFlags`: index of the flags corresponding to the BC's of this equation,
- `m_statesIndex`: vector containing index of the states updated by this equation,
- `m_pMatrixBuilder`: smart pointer to the matrix builder class.

Further work

3D Rayleigh problem

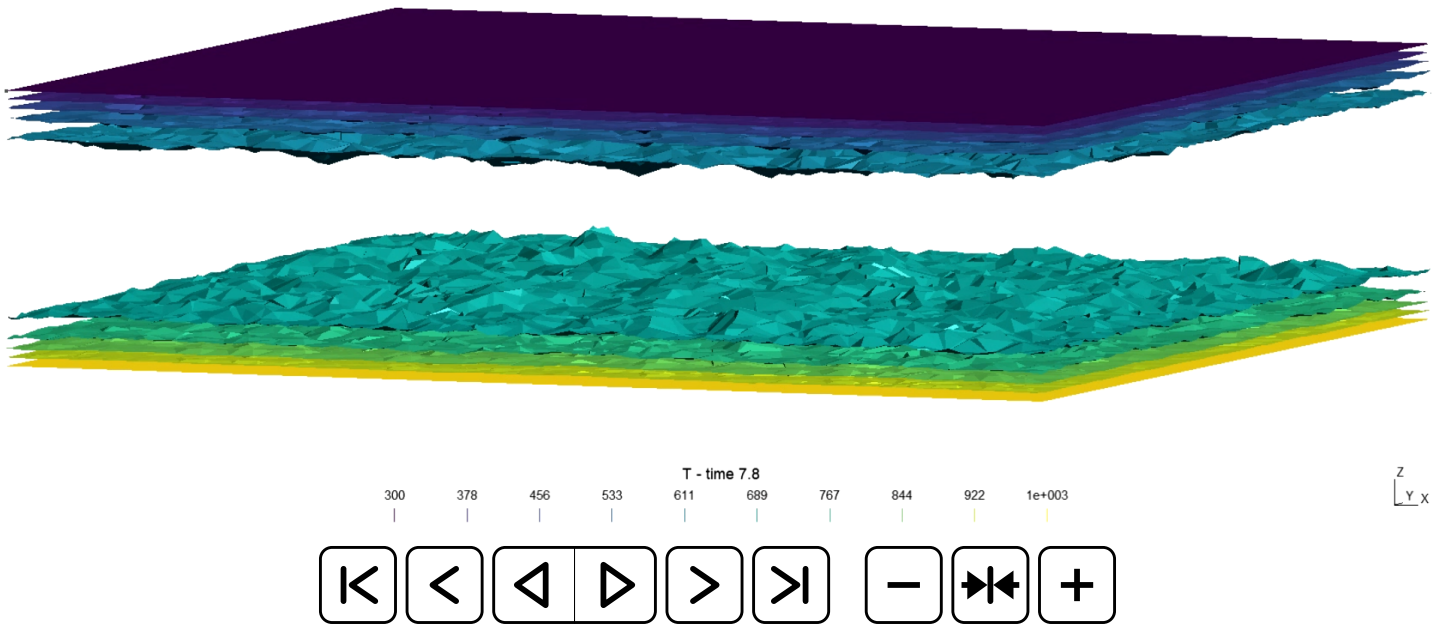
Boussinesq problem using weakly compressible explicit solver:



$T_{\text{min}} = 300\,\text{K},\ T_{\text{max}} = 1000\,\text{K},\ \rho^\star = 1000\,\text{kg}\,\text{m}^{-3},\ \mu = 0.001\,\text{Pa}\,\text{s},$
$k = 0.6\,\text{W}\,\text{m}^{-1}\,\text{K}^{-1},\ K_0 = 2.2 \times 10^7\text{Pa},\ K_0' = 7.6,\ c_v = 1\,\text{J}\,\text{kg}^{-1}\,\text{K}^{-1}.$

3D Rayleigh problem

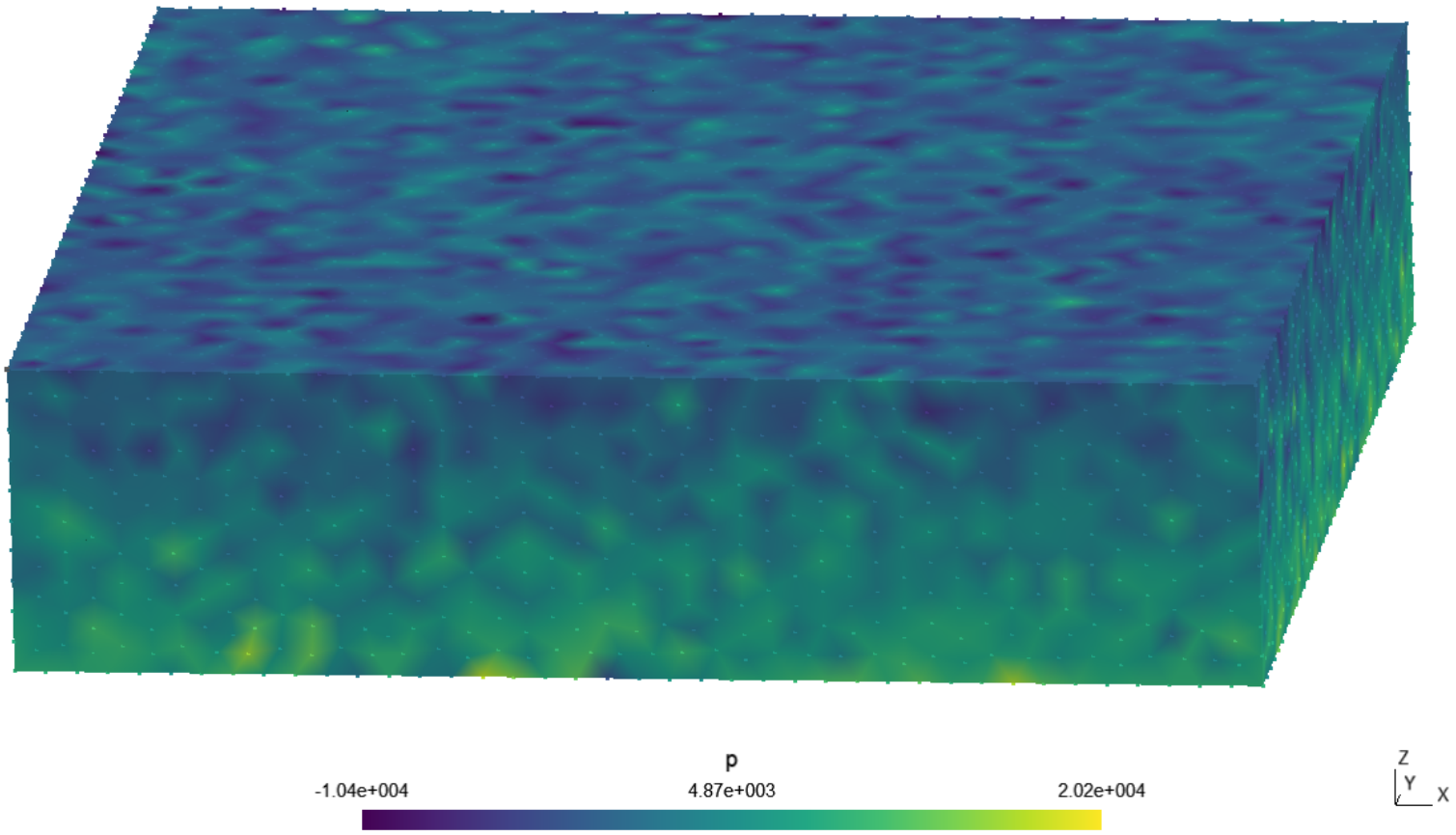
Boussinesq problem using weakly compressible explicit solver:



$T_{\text{min}} = 300\,\text{K},\ T_{\text{max}} = 1000\,\text{K},\ \rho^\star = 1000\,\text{kg}\,\text{m}^{-3},\ \mu = 0.001\,\text{Pa}\,\text{s},$
$k = 0.6\,\text{W}\,\text{m}^{-1}\,\text{K}^{-1},\ K_0 = 2.2 \times 10^7\text{Pa},\ K_0' = 7.6,\ c_v = 1\,\text{J}\,\text{kg}^{-1}\,\text{K}^{-1}.$

3D Rayleigh problem

Boussinesq problem using weakly compressible explicit solver:



Pressure field for the Rayleigh problem.

What happens ?

Continuity equation:

$$\frac{d}{dt}\rho + \rho\vec{\nabla}\cdot\vec{v} = 0$$

Unstabilized equation:

$$\mathbf{M}_l\boldsymbol{\rho}^{n+1} = \mathbf{M}_l\boldsymbol{\rho}^{n} - \Delta t\mathbf{D}_\rho\mathbf{v}$$

- works well when the problem is static,
- meaningless pressure field when fluid moves.

Stabilized equation:

$$\mathbf{M}_l\boldsymbol{\rho}^{n+1} = \mathbf{M}_c\boldsymbol{\rho}^{n} - \Delta t\mathbf{D}_\rho\mathbf{v}$$

- nodes moves a little in a static problem,
- good pressure field when the fluid is in motion.

Further work

- find a way to enable the stabilization when the problem is not static anymore,
- or find a better stabilization,
- work with R. Falla about remeshing,
- begin implementing other temperature dependant material parameter,
- begin implementation of melting.